Polymorphism

- Polymorphism is an object-oriented concept that allows us to create versatile software designs

Binding

- Consider the following method invocation:
- obj.doIt();
- At some point, this invocation is bound to the definition of the method that it invokes
- If this binding occurred at compile time, then that line of code would call the same method every time
- However, Java defers method binding until run time -- this is called **dynamic binding or late binding**

Polymorphism

- The term polymorphism literally means "**having many forms**"
- A **polymorphic reference** is a variable that can refer to different types of objects at different points in time
- The method called through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

Example

```
public abstract class Animal  {  // class is abstract

private String name;

   public Animal(String nm)  {    // constructor method
              name=nm;
      }

   public String getName()    {    // regular method
              return (name);
     }

  public abstract void speak(); // abstract method - note no {}

}
```

- Three subclasses (Cow, Dog and Snake) each having their own speak() method.

Example - Late Method Binding

```
public class AnimalReference
{
  public static void main(String args[]) {
    Animal ref;         // set up var for Animal abstract class

    Cow aCow = new Cow("Bossy"); // makes specific objects
    Dog aDog = new Dog("Rover"); // from the subclasses
    Snake aSnake = new Snake("Ernie");

  // now reference each as an Animal
    ref = aCow; ref.speak();
    ref = aDog; ref.speak();    // resolve references in run-time
    ref = aSnake; ref.speak();
  }
}
```

Example - Array of Objects

```
public class AnimalArray
{
  public static void main(String args[]) {
     Animal ref[] = new Animal[3]; // assign space for array

    Cow aCow = new Cow("Bossy");  // makes specific objects
    Dog aDog = new Dog("Rover");
    Snake aSnake = new Snake("Earnie");

    // now put them in an array
    ref[0] = aCow; ref[1] = aDog; ref[2] = aSnake;

    // now demo dynamic method binding
    for (int x=0;x<3;++x) { ref[x].speak(); }
  }
}
```

Casting Objects

```
Dog doggy = (Dog) ref[x]; //cast current instance to
                          // subclass
doggy.someDogOnlyMethod();
```

Casting an individual instance to its subclass form, one can refer to any property or method

Inheritance and References

```
class C1 { ... }
class C2 extends C1 { ... }
class C3 { ... }
```

```
C1 x;          // x can store a reference to an object of C1 or
               // an object of any subclass of C1.
```

```
// in some other place
C1 o1 = new C1();
C2 o2 = new C2();
C3 o3 = new C3();
```

Inheritance and References (cont.)

```
class C1 { ... }
class C2 extends C1 { ... }
class C3 { ... }

C1 o1 = new C1();       C2 o2 = new C2();       C3 o3 = new C3();

o1 = o2;          Automatically saved (Widening Conversion)

o2 = (C2) o1;   We need explicit type casting (Narrowing Conversion)
                                    if o1 holds C2 object, this is okay;
                                    But, if o1 holds C1 object ➔ run-time error

o1 = o3;          ILLEGAL (There is no inheritance relation between C1 and C3)
o1 = (C1) o3;   ILLEGAL (There is no inheritance relation between C1 and C3)
```

Inheritance and References (cont.)

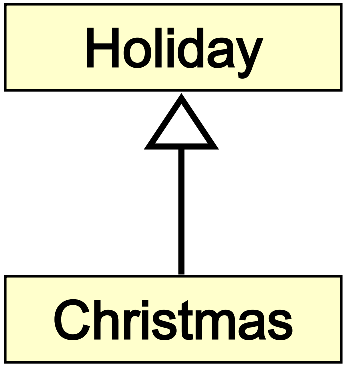
- **Assigning a child reference to a parent reference** is a **widening** conversion, and it can be performed by simple assignment.
    - o1 = o2; ref = aCow;
- **Assigning a parent reference to a child reference** is a **narrowing** conversion, and it must be done with an explicit type cast operation.
    - o2 = (C2) o1;
    - If that parent reference (o1) does not point to a child object (o2) ➔ run-time error
- Since Object class is the ancestor of all classes in Java, we can store any type of reference in an Object reference variable

Inheritance and References (cont.)

```
class C1 { ... }
class C2 extends C1 { ... }

// in some other place
C1 o1 = new C1();  C2 o2 = new C2();
Object o3;

o3 = o1;       Automatically saved (Widening Conversion)

o3 = o2;       Automatically saved (Widening Conversion)
```

References and Inheritance

- An object reference can refer to an object of any class related to it by inheritance
- For example, if Holiday is the superclass of Christmas, then a Holiday reference could be used to refer to a Christmas object



```
Holiday day;
day = new Christmas();
```

References and Inheritance

- These type compatibility rules are just an extension of the is-a relationship established by inheritance
- Assigning a Christmas object to a Holiday reference is fine because **Christmas is-a holiday**
- Assigning a child object to a parent reference can be performed by simple assignment
- Assigning an parent object to a child reference can be done also, but must be done with a cast
- After all, Christmas is a holiday but **not all holidays are Christmas**

Polymorphism via Inheritance

- Now suppose the Holiday class has a method called celebrate, and Christmas overrides it
- What method is invoked by the following?

```
day.celebrate();
```

- The **type of the object** being referenced, **not the reference type**, determines which method is invoked
- If day refers to a Holiday object, it invokes the Holiday version of celebrate; if it refers to a Christmas object, it invokes that version

Polymorphism via Inheritance

- Note that the **compiler restricts invocations based on the type of the reference**
- So if Christmas had a method called getTree that Holiday didn't have, the following would cause a compiler error:

```
day.getTree();  // compiler error
```

- Remember, the compiler doesn't "know" which type of holiday is being referenced
- A cast can be used to allow the call:
- ((Christmas)day).getTree();

Quick Check

If `MusicPlayer` is the parent of `CDPlayer`, are the following assignments valid?

```
MusicPlayer mplayer = new CDPlayer();
```

```
CDPlayer cdplayer = new MusicPlayer();
```

Quick Check

If `MusicPlayer` is the parent of `CDPlayer`, are the following assignments valid?

```
MusicPlayer mplayer = new CDPlayer();
```

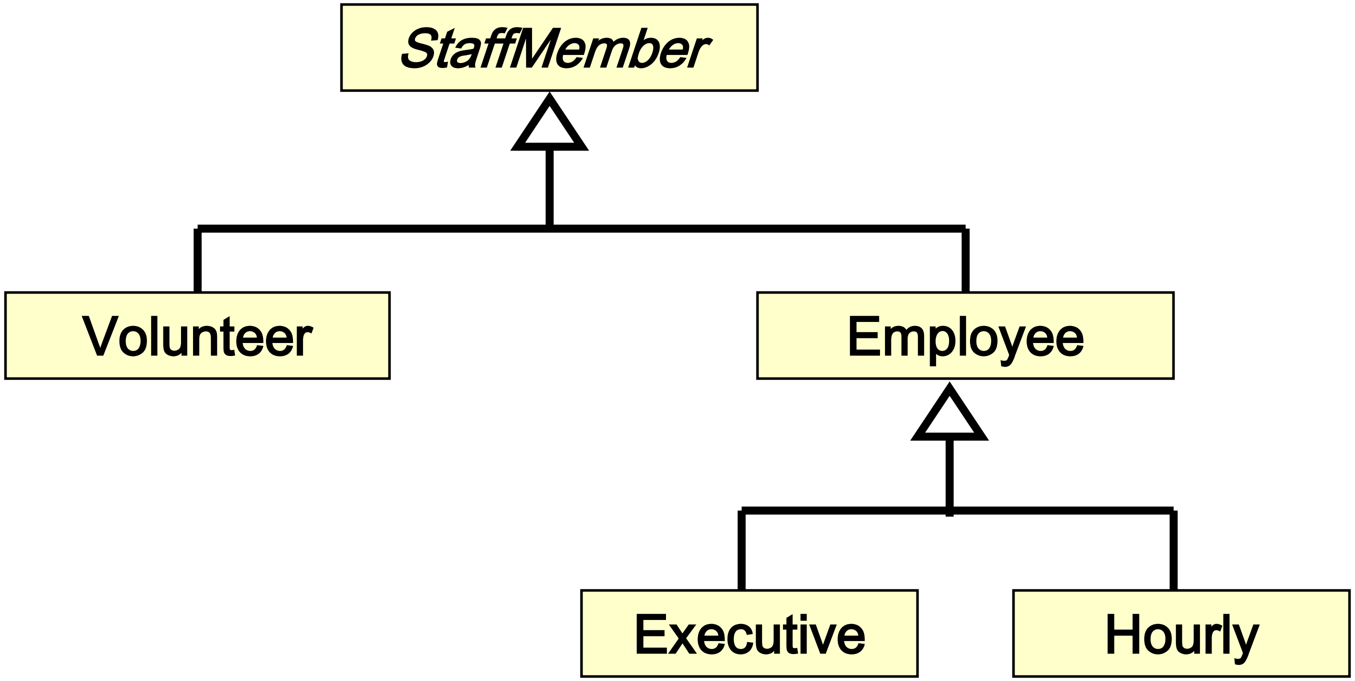
Yes, because a `CDPlayer` is-a `MusicPlayer`

```
CDPlayer cdplayer = new MusicPlayer();
```

No, you'd have to use a cast (and you shouldn't knowingly assign a super class object to a subclass reference)

Polymorphism via Inheritance

- Consider the following class hierarchy:



Polymorphism via Inheritance

- Let's look at an example that pays a set of diverse employees using a polymorphic method
- See Firm.java
- See Staff.java
- See StaffMember.java
- See Volunteer.java
- See Employee.java
- See Executive.java
- See Hourly.java

```
//********************************************************************
//  Firm.java       Author: Lewis/Loftus
//
//  Demonstrates polymorphism via inheritance.
//********************************************************************

public class Firm
{
   //-----------------------------------------------------------------
   //  Creates a staff of employees for a firm and pays them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Staff personnel = new Staff();

      personnel.payday();
   }
}
```

Output

```
Name: Sam
Address: 123 Main Line
Phone: 555-0469
Social Security Number: 123-45-6789
Paid: 2923.07
-----------------------------------
Name: Carla
Address: 456 Off Line
Phone: 555-0101
Social Security Number: 987-65-4321
Paid: 1246.15
-----------------------------------
Name: Woody
Address: 789 Off Rocker
Phone: 555-0000
Social Security Number: 010-20-3040
Paid: 1169.23
-----------------------------------
```

Output (continued)

```
Name: Diane
Address: 678 Fifth Ave.
Phone: 555-0690
Social Security Number: 958-47-3625
Current hours: 40
Paid: 422.0
-----------------------------------
Name: Norm
Address: 987 Suds Blvd.
Phone: 555-8374
Thanks!
-----------------------------------
Name: Cliff
Address: 321 Duds Lane
Phone: 555-7282
Thanks!
-----------------------------------
```

```java
//********************************************************************
//  Staff.java       Author: Lewis/Loftus
//
//  Represents the personnel staff of a particular business.
//********************************************************************

public class Staff
{
   private StaffMember[] staffList;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the list of staff members.
   //-----------------------------------------------------------------
   public Staff ()
   {
      staffList = new StaffMember[6];
```

continue

continue

```
      staffList[0] = new Executive ("Sam", "123 Main Line",
         "555-0469", "123-45-6789", 2423.07);

      staffList[1] = new Employee ("Carla", "456 Off Line",
         "555-0101", "987-65-4321", 1246.15);
      staffList[2] = new Employee ("Woody", "789 Off Rocker",
         "555-0000", "010-20-3040", 1169.23);

      staffList[3] = new Hourly ("Diane", "678 Fifth Ave.",
         "555-0690", "958-47-3625", 10.55);

      staffList[4] = new Volunteer ("Norm", "987 Suds Blvd.",
         "555-8374");
      staffList[5] = new Volunteer ("Cliff", "321 Duds Lane",
         "555-7282");

      ((Executive)staffList[0]).awardBonus (500.00);

      ((Hourly)staffList[3]).addHours (40);
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Pays all staff members.
   //-----------------------------------------------------------------
   public void payday ()
   {
      double amount;

      for (int count=0; count < staffList.length; count++)
      {
         System.out.println (staffList[count]);

         amount = staffList[count].pay();  // polymorphic

         if (amount == 0.0)
            System.out.println ("Thanks!");
         else
            System.out.println ("Paid: " + amount);

         System.out.println ("-----------------------------------");
      }
   }
}
```

```
//********************************************************************
//  StaffMember.java       Author: Lewis/Loftus
//
//  Represents a generic staff member.
//********************************************************************

abstract public class StaffMember
{
   protected String name;
   protected String address;
   protected String phone;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this staff member using the specified
   //  information.
   //-----------------------------------------------------------------
   public StaffMember (String eName, String eAddress, String ePhone)
   {
      name = eName;
      address = eAddress;
      phone = ePhone;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Returns a string including the basic employee information.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = "Name: " + name + "\n";

      result += "Address: " + address + "\n";
      result += "Phone: " + phone;

      return result;
   }

   //-----------------------------------------------------------------
   //  Derived classes must define the pay method for each type of
   //  employee.
   //-----------------------------------------------------------------
   public abstract double pay();
}
```

```java
//********************************************************************
//  Volunteer.java       Author: Lewis/Loftus
//
//  Represents a staff member that works as a volunteer.
//********************************************************************

public class Volunteer extends StaffMember
{
   //-----------------------------------------------------------------
   //  Constructor: Sets up this volunteer using the specified
   //  information.
   //-----------------------------------------------------------------
   public Volunteer (String eName, String eAddress, String ePhone)
   {
      super (eName, eAddress, ePhone);
   }

   //-----------------------------------------------------------------
   //  Returns a zero pay value for this volunteer.
   //-----------------------------------------------------------------
   public double pay()
   {
      return 0.0;
   }
}
```

```
//********************************************************************
//  Employee.java       Author: Lewis/Loftus
//
//  Represents a general paid employee.
//********************************************************************

public class Employee extends StaffMember
{
   protected String socialSecurityNumber;
   protected double payRate;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this employee with the specified
   //  information.
   //-----------------------------------------------------------------
   public Employee (String eName, String eAddress, String ePhone,
                    String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone);

      socialSecurityNumber = socSecNumber;
      payRate = rate;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Returns information about an employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nSocial Security Number: " + socialSecurityNumber;

      return result;
   }

   //-----------------------------------------------------------------
   //  Returns the pay rate for this employee.
   //-----------------------------------------------------------------
   public double pay()
   {
      return payRate;
   }
}
```

```java
//********************************************************************
//  Executive.java       Author: Lewis/Loftus
//
//  Represents an executive staff member, who can earn a bonus.
//********************************************************************

public class Executive extends Employee
{
   private double bonus;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this executive with the specified
   //  information.
   //-----------------------------------------------------------------
   public Executive (String eName, String eAddress, String ePhone,
                     String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);

      bonus = 0;  // bonus has yet to be awarded
   }
```

continue

continue

```
   //----------------------------------------------------------------
   //  Awards the specified bonus to this executive.
   //----------------------------------------------------------------
   public void awardBonus (double execBonus)
   {
      bonus = execBonus;
   }

   //----------------------------------------------------------------
   //  Computes and returns the pay for an executive, which is the
   //  regular employee payment plus a one-time bonus.
   //----------------------------------------------------------------
   public double pay()
   {
      double payment = super.pay() + bonus;

      bonus = 0;

      return payment;
   }
}
```

```
//********************************************************************
//  Hourly.java       Author: Lewis/Loftus
//
//  Represents an employee that gets paid by the hour.
//********************************************************************

public class Hourly extends Employee
{
   private int hoursWorked;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this hourly employee using the specified
   //  information.
   //-----------------------------------------------------------------
   public Hourly (String eName, String eAddress, String ePhone,
                  String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);

      hoursWorked = 0;
   }
```

continue

continue

```
   //-------------------------------------------------------------------
   //  Adds the specified number of hours to this employee's
   //  accumulated hours.
   //-------------------------------------------------------------------
   public void addHours (int moreHours)
   {
      hoursWorked += moreHours;
   }

   //-------------------------------------------------------------------
   //  Computes and returns the pay for this hourly employee.
   //-------------------------------------------------------------------
   public double pay()
   {
      double payment = payRate * hoursWorked;

      hoursWorked = 0;

      return payment;
   }
```

continue

continue

```
   //----------------------------------------------------------------
   //  Returns information about this hourly employee as a string.
   //----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nCurrent hours: " + hoursWorked;

      return result;
   }
}
```

Polymorphism via Interfaces

- Interfaces can be used to set up polymorphic references as well
- Suppose we declare an interface called Speaker as follows:

```
public interface Speaker
{
   public void speak();
   public void announce (String str);
}
```

Polymorphism via Interfaces

- An **interface name can be used as the type of an object reference variable**:

```
Speaker current;
```

- The current reference can be used to point to any object of any class that implements the Speaker interface
- **The version of speak invoked by the following line depends on the type of object that current is referencing**:

```
current.speak();
```

Polymorphism via Interfaces

- Now suppose two classes, Philosopher and Dog, both implement the Speaker interface, providing distinct versions of the speak method

- In the following code, the first call to speak invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```

Polymorphism via Interfaces

- As with class reference types, **the compiler will restrict invocations to methods in the interface**

- For example, even if Philosopher also had a method called pontificate, the following would still cause a compiler error:

```
Speaker special = new Philospher();
special.pontificate();  // compiler error
```

- Remember, the compiler bases its rulings on the type of the reference

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();
Philosopher second = new Philosopher();
second.pontificate();
first = second;
```

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();
Philosopher second = new Philosopher();
second.pontificate();
first = second;
```

Yes, all assignments and method calls are valid as written